COMP5338 Neo4j Project Report

Albert Huang 530215563 qhua0468

1 Introduction

This project is the fourth assignment of COMP5338: Advanced Data Models. It is set to showcase author's mastery on Neo4j, more specially, the ability to build a browser guide, load data, build graph model, find solutions using Neo4j queries and analyses the performance of workloads.

A data set is given. The data set is from ***OpenFlights Data***, containing 4 *dat* files: *airports.dat*, *airlines.dat*, *planes.dat* and *routes.dat*, with information respectively.

The desired outcome of the project is to complete all tasks while satisfy all requirements from *COMP5338_2024_Neo4j_Assignment_4.pdf* given on the Assignment page in the unit. It is expected that after the completion of the project, author will obtain a higher overall understanding of Neo4j and build a stronger skill on operating Neo4j.

2 Graph Model

2.1 Preparing Environment

1. Put all 4 *dat* files in the *import* folder of the project's DBMS in Neo4j. Create a new database in that DBMS.
2. Under the same folder of the browser guide (*browser_guide_530215563.html*), open terminal/powershell/cmd, and use code: python -m http.server 8000 to host a local server.
3. Open Neo4j browser in the project's DMBS. Switch to corresponding database. Use command: :play http://localhost:8000/browser_guide_530215563.html to load and use the browser guide.

2.2 Model Description

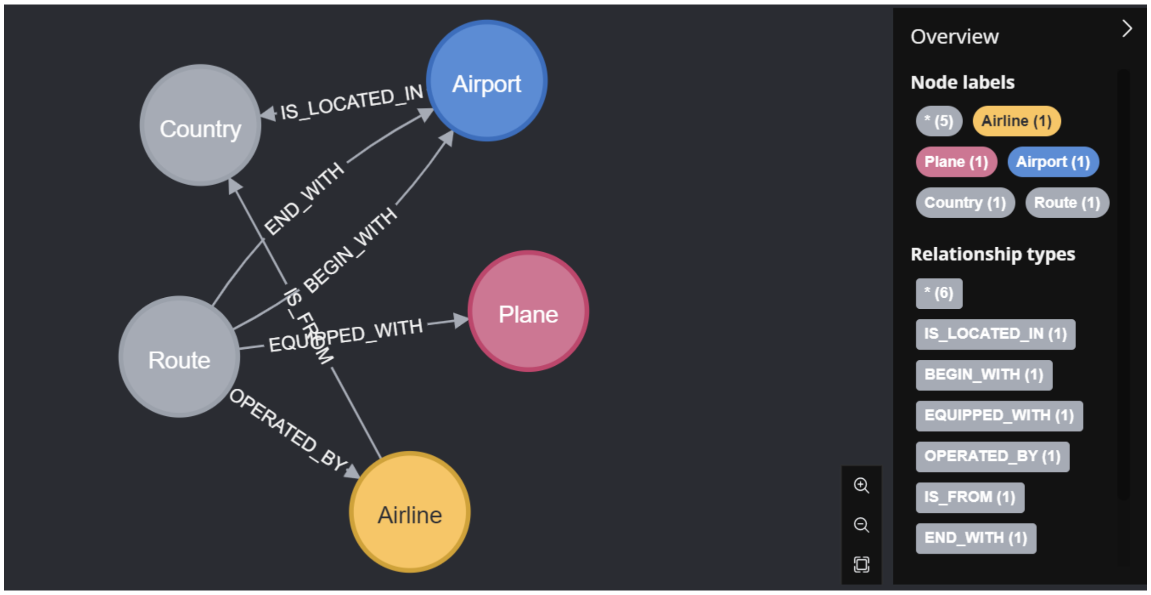


Figure 1: Graph Schema

In my graph model, there are:

80,737 nodes, in 5 types of node labels:

- **Airport** Each Airport node contains all data of an airport in *airports.dat* (one row). A Airport node has following properties:
  - **id** Corresponding to **Airport ID** in the original column. It is converted to integer.
  - **name** Corresponding to **Name** in the original column.
  - **city** Corresponding to **City** in the original column.
  - **iata** Corresponding to **IATA** in the original column.
  - **icao** Corresponding to **ICAO** in the original column.
  - **latitude** Corresponding to **Latitude** in the original column. It is converted to float.
  - **longitude** Corresponding to **Longitude** in the original column. It is converted to float.
- **Airline** Each Airline node contains all data of an airline in *airlines.dat* (one row). A Airline node has following properties:
  - **id** Corresponding to **Airline ID** in the original column. It is converted to integer.
  - **name** Corresponding to **Name** in the original column.
  - **alias** Corresponding to **Alias** in the original column.
  - **iata** Corresponding to **IATAICAO** in the original column.
  - **icao** Corresponding to **ICAO** in the original column.
  - **callsign** Corresponding to **Callsign** in the original column.

  - **active** Corresponding to **Active** in the original column.
- **Plane** Each Plane node contains all data of a plane in *planes.dat* (one row). A Plane node has following properties:
  - **name** Corresponding to **Name** in the original column.
  - **iata** Corresponding to **IATA code** in the original column.
  - **icao** Corresponding to **ICAO code** in the original column.
- **Route** Each Route node contains all data of a route in *routes.dat* (one row). All routes being loaded into nodes here are those valid. Meaning each route has a source airport id,a destination airport id and airline id that can be referenced back to *airlines.dat* and *airports.dat*. A Route node has following properties:
  - **airline** Corresponding to **Airline** in the original column.
  - **airline_id** Corresponding to **Airline ID** in the original column.
  - **source_airport** Corresponding to **Source airport** in the original column.
  - **source_id** Corresponding to **Source airport ID** in the original column.
  - **destiation_airport** Corresponding to **Destination airport** in the original column.
  - **destination_id** Corresponding to **Destination airport ID** in the original column.
  - **code_share** Corresponding to **Codeshare** in the original column.
  - **stops** Corresponding to **Stops Equipment** in the original column.
  - **equipment** Corresponding to **Equipment** in the original column.
- **Country** This node contains the names of every country mentioned in the data set. It has one property: **name**.

There are 7698 Airport nodes, 6160 Airline nodes (empty row deleted), 246 Plane nodes, and 66316 Route nodes.

290,610 relationships, in 6 relationship types:

- **BEGIN_WITH** This relationship represents a route begin with an airport.
- **END_WITH** This relationship represents a route end with an airport.
- **EQUIPPED_WITH** This relationship represents a route is equipped with a plane.
- **IS_FROM** This relationship represents an airline is from a country.
- **IS_LOCATED_IN** This relationship represents an airport is located in a country.
- **OPERATED_BY** This relationship represents a route is operated by an airline.

2.3 Design Justification

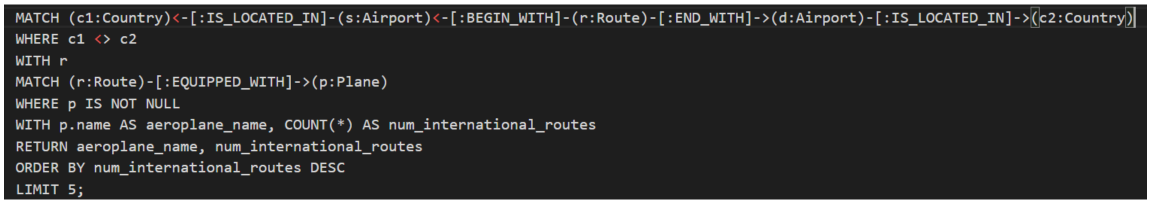
When designing the graph model, node labels are the first to be considered. It is a straightforward thinking that airport, airline, plane should have its own node label. When it comes to route, it is initially being designed as a relationship between a source airport and a destination airport in a route. However, since route is the only entity that have connection to other three entities(airport, airline, plane), it is suitable that route should be made a node label. Country is separated from airport and airline and be made a node label for querying convenience in later part.

For relationship, route has 4 kind of relationships according to its connection with other entities. Each of these relationship represents a property in route that connect to a property in other entity. Lastly, relationship between airline, airport and country are considered as each airport and airline is located in a country.

What also needs to be mentioned is that constraints are being made to ensure the uniqueness of some properties. So far no index is needed for data loading and graph building as the process complete in a short time.

3 Workload Implementation

3.1 NW1

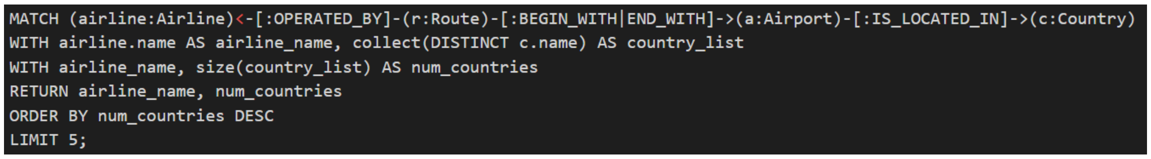


```
MATCH (c1:Country)<-[:IS_LOCATED_IN]-(s:Airport)<-[:BEGIN_WITH]-(r:Route)-[:END_WITH]->(d:Airport)-[:IS_LOCATED_IN]->(c2:Country)
WHERE c1 <> c2
WITH r
MATCH (r:Route)-[:EQUIPPED_WITH]->(p:Plane)
WHERE p IS NOT NULL
WITH p.name AS aeroplane_name, COUNT(*) AS num_international_routes
RETURN aeroplane_name, num_international_routes
ORDER BY num_international_routes DESC
LIMIT 5;
```

Figure 2: NW1

1. In this implementation, it first matches entire routes: country-airport-route-airport-country. (line 1)
2. Then filter in those airports in the different countries. (line 2)
3. And use these international routes to match their equipped planes. (line 3 to 5)
4. Next group planes by its name and count the amount of routes in each group to get the number of international routes of each plane type. (line 6)
5. Lastly, return the top five planes with number of international routes as required. (line 7 to 9)

3.2 NW2

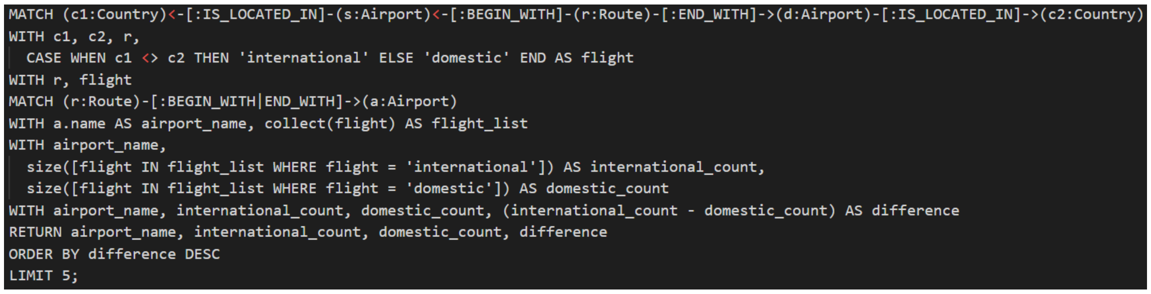


```
MATCH (airline:Airline)<-[:OPERATED_BY]-(r:Route)-[:BEGIN_WITH|END_WITH]->(a:Airport)-[:IS_LOCATED_IN]->(c:Country)
WITH airline.name AS airline_name, collect(DISTINCT c.name) AS country_list
WITH airline_name, size(country_list) AS num_countries
RETURN airline_name, num_countries
ORDER BY num_countries DESC
LIMIT 5;
```

Figure 3: NW2

1. In this implementation, it starts with matching entire routes with operating airlines, source and destination airport with countries its in. (line 1)
2. Then it use airline name for grouping, collect distinct countries in list for each airline. (line 2)
3. Next, it calculate the size of each list as number of distinct countries. (line 3)
4. Lastly, return the output as required. (line 4 to 6)

3.3 NW3

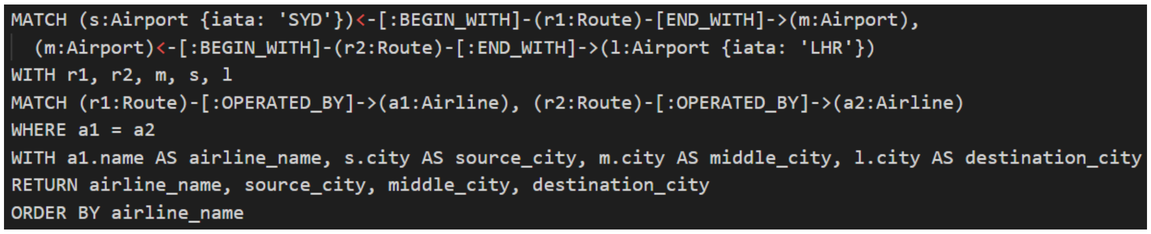


```
MATCH (c1:Country)<-[:IS_LOCATED_IN]-(s:Airport)<-[:BEGIN_WITH]-(r:Route)-[:END_WITH]->(d:Airport)-[:IS_LOCATED_IN]->(c2:Country)
WITH c1, c2, r,
  CASE WHEN c1 <> c2 THEN 'international' ELSE 'domestic' END AS flight
WITH r, flight
MATCH (r:Route)-[:BEGIN_WITH|END_WITH]->(a:Airport)
WITH a.name AS airport_name, collect(flight) AS flight_list
WITH airport_name,
  size([flight IN flight_list WHERE flight = 'international']) AS international_count,
  size([flight IN flight_list WHERE flight = 'domestic']) AS domestic_count
WITH airport_name, international_count, domestic_count, (international_count - domestic_count) AS difference
RETURN airport_name, international_count, domestic_count, difference
ORDER BY difference DESC
LIMIT 5;
```

Figure 4: NW3

1. In this implementation, it first matches entire chain of routes like in NW1 did. (line 1)
2. Then, it classified each route into ”international” or ”domestic” by the two countries source airport and destination airport are in. (line 2 to 4)
3. Next, it matches route with their related airports again. (line 5)
4. After that, it group the airport by its name and collect route type for every route in a airport as a flight list. (line 6)
5. Every type of flights in a flight list is being counted. (line 7 to 9)
6. Difference of number international flights and domestic flights in each airport are calculated. (line 10)
7. Lastly, return the output as required. (line 11 to 13)

3.4 NW4

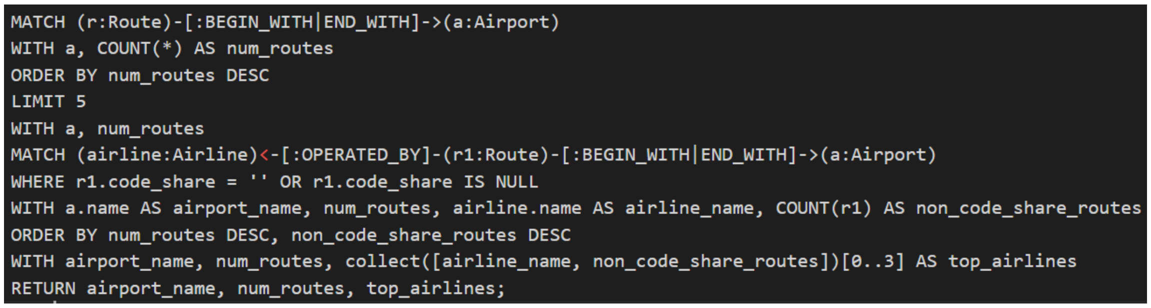


```
MATCH (s:Airport {iata: 'SYD'})<-[:BEGIN_WITH]-(r1:Route)-[END_WITH]->(m:Airport),
  (m:Airport)<-[:BEGIN_WITH]-(r2:Route)-[:END_WITH]->(l:Airport {iata: 'LHR'})
WITH r1, r2, m, s, l
MATCH (r1:Route)-[:OPERATED_BY]->(a1:Airline), (r2:Route)-[:OPERATED_BY]->(a2:Airline)
WHERE a1 = a2
WITH a1.name AS airline_name, s.city AS source_city, m.city AS middle_city, l.city AS destination_city
RETURN airline_name, source_city, middle_city, destination_city
ORDER BY airline_name
```

Figure 5: NW4

1. This implementation first matches pairs of routes that start with SYD, end with a airport and starts from that airprot, end with LHR as destination airport. (line 1 to 2)
2. Then it checks if these each pair of routes is operated by the same airline. (line 3 to 4)
3. Next, it return the airline name, source city, middle city, and destination city as required. (line 5 to 7)

3.5 NW5



```
MATCH (r:Route)-[:BEGIN_WITH|END_WITH]->(a:Airport)
WITH a, COUNT(*) AS num_routes
ORDER BY num_routes DESC
LIMIT 5
WITH a, num_routes
MATCH (airline:Airline)<-[:OPERATED_BY]-(r1:Route)-[:BEGIN_WITH|END_WITH]->(a:Airport)
WHERE r1.code_share = '' OR r1.code_share IS NULL
WITH a.name AS airport_name, num_routes, airline.name AS airline_name, COUNT(r1) AS non_code_share_routes
ORDER BY num_routes DESC, non_code_share_routes DESC
WITH airport_name, num_routes, collect([airline_name, non_code_share_routes])[0..3] AS top_airlines
RETURN airport_name, num_routes, top_airlines;
```

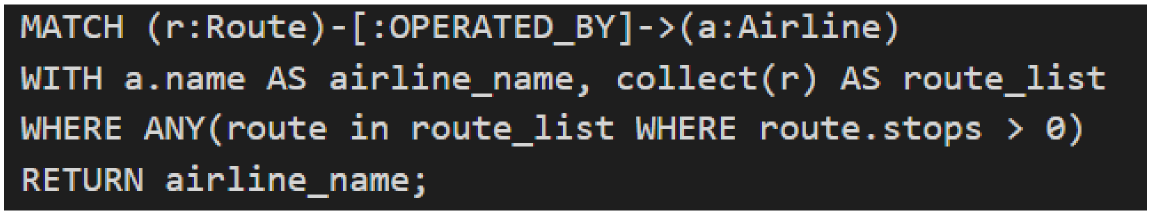
Figure 6: NW5

1. This implementation first matches every routes with its related airports. (line 1)
2. Then group airports for every distinct value, and for each airport, calculate the number of routes served. (line 2)
3. Pick the top 5 airports with the highest number of routes served. (line 3 to 4)
4. Next, matches all routes operated by non-code-share airlines for each top 5 airports. (line 5 to 7)
5. For each top 5 airports, group routes by operating airline name and calculate the number of routes under each airline. (line 8)
6. After that, pick out the top 3 airlines with the highest number of non-code-share routes under each top 5 airport. (line 9 to 10)
7. Lastly, return the output according to requirements. (line 11)

4 Query Design and Implementation

4.1 DW1

ANY is set to be used in the beginning. To use **ANY**, there should be a list. Then a list of routes is considered. Lastly, routes with at least a stop is set to be the value here.



```
MATCH (r:Route)-[:OPERATED_BY]->(a:Airline)
WITH a.name AS airline_name, collect(r) AS route_list
WHERE ANY(route in route_list WHERE route.stops > 0)
RETURN airline_name;
```

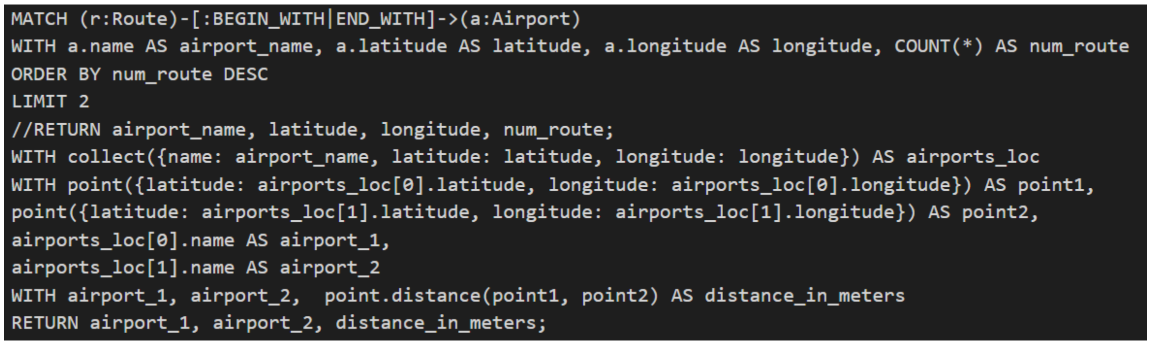
Figure 7: DW1

The purpose of this query is to find airlines who operate at least one route with stops.

ANY is used to get airline with a least one route with one or more than one stops in the route list.

4.2 DW2

For spatial function, **point** is essential here, and to give a meaning to the query, **point.distance** is also selected. And **collect** as a list function organically appeared when needed in the query. In here, goal was first set, then query was designed according to the goal.



```
MATCH (r:Route)-[:BEGIN_WITH|END_WITH]->(a:Airport)
WITH a.name AS airport_name, a.latitude AS latitude, a.longitude AS longitude, COUNT(*) AS num_route
ORDER BY num_route DESC
LIMIT 2
//RETURN airport_name, latitude, longitude, num_route;
WITH collect({name: airport_name, latitude: latitude, longitude: longitude}) AS airports_loc
WITH point({latitude: airports_loc[0].latitude, longitude: airports_loc[0].longitude}) AS point1,
point({latitude: airports_loc[1].latitude, longitude: airports_loc[1].longitude}) AS point2,
airports_loc[0].name AS airport_1,
airports_loc[1].name AS airport_2
WITH airport_1, airport_2,  point.distance(point1, point2) AS distance_in_meters
RETURN airport_1, airport_2, distance_in_meters;
```

Figure 8: DW2

This query is used to find the direct distance between the two airports with highest number of serving flights/routes.

point is used to organized the coordinates of two selected airports into the form needed for later calculation. **point.distance** is used to get the distance between to two points. **collect** is used to collect data needed for spatial function and stored them in a suitable way.

5 Performance Observation

5.1 NW1

5.1.1 Execution Plan

1. **NodeBYLabelScan On Country:** No index is used here. All nodes with Country label is scanned.
2. **Expand On IS_LOCATED_IN:** This step expands from Country to Airport using the IS_LOCATED_IN relationship.
3. **Filter On Airport:** This step filters airports not fitting the matching conditions.
4. **Expand On BEGIN_WITH:** Like step 2, it continues to expand to Route node using BEGIN_WITH relationship.
5. **Filter On Route:** Like step 3, it filters routes which not fitting conditions.
6. **Expand On EQUIPPED_WITH:** Expand to Plane node using EQUIPPED_WITH relationship.
7. **Filter On Plane:** Filter out Plane node to check validation.
8. **Expand On END_WITH** Expand to destination airport using END_WITH relationship.
9. **Filter On Country:** Compare the countries of source airport and destination airport to filter out non-international routes.

10. **EagerAggregation:** This step aggregates results, counting the number of international routes we needed here.
11. **Top:** Organize result by sorting and limiting to get the top 5 we needed.
12. **ProduceResults:** Producing result in desired fashion.

5.1.2 Improvement

In my implementation, no index is used as the runtime and resource spent still in a manageable range for average PCs. However, there is room for potential performance improvement:

- **Adding Indexes:** Create indexes on Country, Airport, Route, nodes, and Plane.name could improve performance as these are used in this query. The expected outcome of adding these indexes is the reduction of runtime on scanning and lower expanding costs.

5.2 NW2

5.2.1 Execution Plan

1. **NodeByLabelScan On Country:** It starts with carrying out a node label scan on Counrty, with no indexes used. All nodes is being scanned.
2. **Cache Properties:** Cache Country properties as they will be frequently accessed and used. This would improve performance.
3. **Expand On IS_LOCATED_IN:** This step expand to airport nodes using IS_LOCATED_IN relationship.
4. **Expand on BEGIN_WITH / END_WITH:** Here, it expand to Airport nodes using BEGIN_WITH or END_WITH relationships.
5. **Filter on Route:** Filter out routes that are not fitting the conditions.
6. **Expand On OPERATED_BY:** Expand route nodes to Airline nodes via OPERATED_BY relationship.
7. **EagerAggregation:** This step aggregates results, collecting every distinct country names to a list for each airline.
8. **Projection:** Projects the number of distinct countries for each airline. It calculates the size of lists to get the numbers.
9. **Top:** Organize output by sorting and limiting. It gets the top 5 results we needed.
10. **ProduceResults:** Produces the results as required.

5.2.2 Improvement

Based on the analysis on execution plan above, no indexes has been adopted and used so far. The current performance is still satisfiable for geneal, however, room for potential improvement can be implemented through the following methods:

- **Create Indexes:** Indexes on the relationship between Country and Airport, Routes could enhance performance as they are related to steps with high hits in the plan.

5.3 NW4

5.3.1 Execution Plan

The execution plan here is a bit special. It begins with two branches in parallel, as there are two chains in matching stage. Then it merge into one mainstream execution flow.

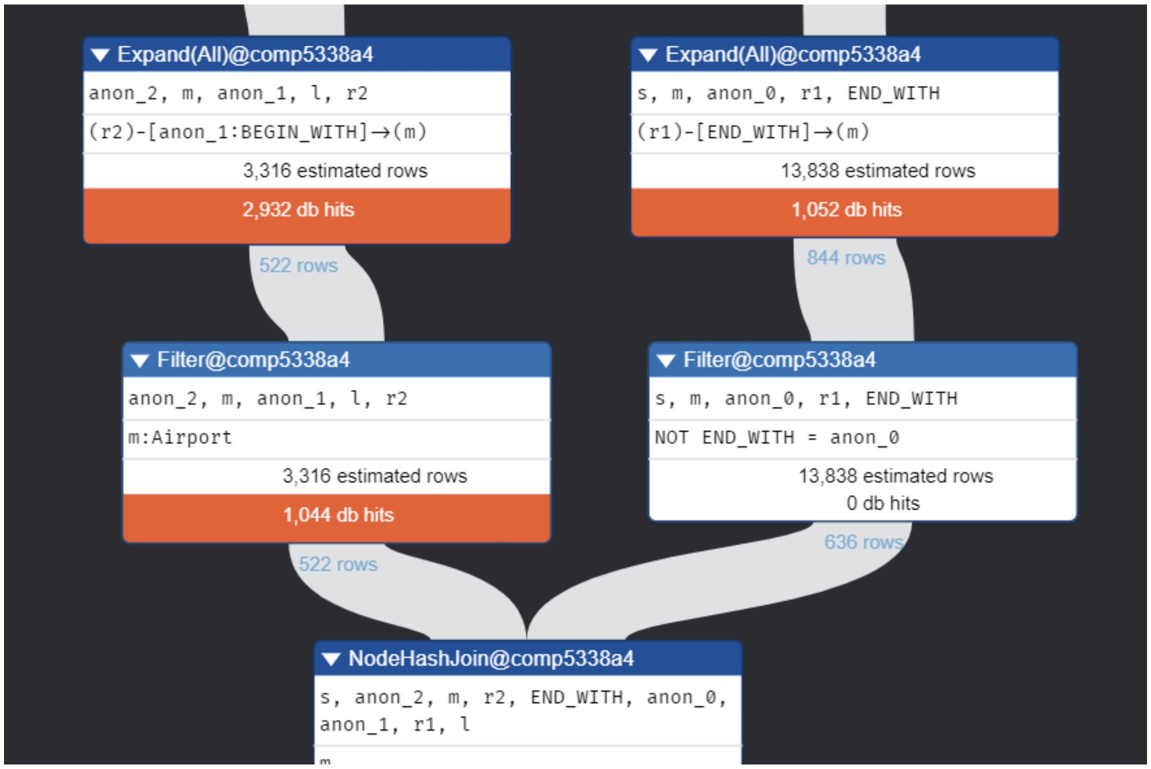


Figure 9: Participial Execution Plan

In details, the plan goes as following:

1. **NodeByLabelScan On Airport:** Since there are two chains in the macthing line, there are two full node label scan on Airport. The two branches has the same type of steps simultaneously.
2. **Filter on Airport:** Use s.iata = 'SYD' and l.iata = 'LHR' to perform filtering on each branches respectively.
3. **Expand On END_WITH & BEGIN_WITH:** Similar to step 2, expand to route nodes using END_WITH & BEGIN_WITH relationships on two branches respectively.
4. **Filter On Route:** Filter out routes not fitting conditions on the two branches.
5. **Expand(All):** For both branches, expand all possible chains that fits the conditions, variables.
6. **Filter On Airport:** This filter make sure for both branches, a middle city m exists.
7. **NodeHashJoin:** This step joins the two branches with middle airport m as the connecting node. In other way, it creates pairs of route 1 chain and route 2 chain using m.

8. **Filter On Airline:** This step filter out airlines that are not the same for the same pair of route 1 and route 2.
9. **Sequence of Expand & Filter:** These steps expand to airlines using OPERATED_BY relationship and filter out ailrines not fitting conditions.
10. **Projection:** Projects all attributes as required.
11. **Sort:** Sort the results by airline names in ascending order.
12. **ProduceResults:** Produces results according to requirements.

5.3.2 Improvement

Considering the size of data used and query complexity here, no indexes is urgently needed, and hence no index usage so far. However, indexes could be added along with other methods of improvements for performance optimizing:

- **Index Usage:** Add an index for iata property on Airport nodes. This is expected to be a great improvement as this property is vital and heavily used in this query. Also, indexes on route nodes for BEGIN_WITH, END_WITH, and OPERATED_BY relationship can bring significant improvement on performance as they are used frequently as well.
- **Querying Structural Optimization:** COnstruct the query in a different way that, it doesn't require two clauses for matching. It could simplified the execution flow and burdens. However, it needs to be experiments to see actually effect.

6 Conclusion

It can be confirmed that the project has been completed according to requirements. All workloads is implemented. Improvements on certain workloads had been implemented for a better performance. The purpose of this project had been successfully served, as author's overall mastery on Neo4j is deeper and enhanced, and the final result of the project is believed to reflect on author's knowledge and skills on Neo4j.